Abstracting over Data Definitions

A bank customer can have three different accounts: a checking account, a savings account, and a line of credit account.

- The customer can withdraw from a checking account any amount that will still leave the minimum balance in the account. The customer can withdraw all money from a savings account. The balance of the credit line represents the amount that the customer already borrowed against the credit line. The customer can withdraw any amount that does not make the balance exceed the credit limit.
- The customer can deposit money to the account in any amount. If the customer deposits more to the credit line than the current balance, the balance will become negative — indicating overpayment.

4.1 Review of Designing Methods for Unions of Classes.

The code in *lab4-banking.bjava* defines the classes that represent this information.

1. Make examples of data for these classes, then make sure you understand the rules for withdrawals.

   Now design the methods that will manage the banking records:

2. Design the method *canWithdraw* that determines whether the customer can withdraw some desired amount.
3. Design the method *makeDeposit* that allows the customer to deposit a given amount of money into the account.
4. Design the method *maxWithdrawal* that computes the maximum that the customer can withdraw from an account.
5. Design the method *moreAvailable* that produces the account that has more money available for withdrawal.

4.2 Abstracting over Data Definitions: Lifting Fields

Save your work and open it again with the file type 'ijava'. Change the language level to Intermediate ProfessorJ.

Look at the code and identify all places where the code repeats — the opportunity for abstraction.

Lift the common fields to an abstract class *ABanking*. Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

4.3 Abstracting over Data Definitions: Lifting Methods

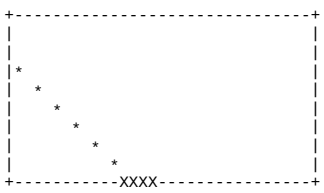
For each method that is defined in all three classes decide to which category it belongs:

1. The method bodies in the different classes are all different, and so the method has to be declared as *abstract* in the *abstract* class.
2. The method bodies are the same in all classes and it can be implemented completely in the *abstract* class.
3. The methods look very similar, but each produces a different variant of the union — therefore it cannot be lifted to the *super* class.
4. The method bodies are the same for two of the classes, but are different in one class — therefore we can define the common body in the *abstract* class and override it in only one derived class.

Now, lift the methods that can be lifted and run all tests again.

Part 2: Designing the Pong Game

A Game of Pong



Rules:

A ball starts at a random height on the left and falls from the left side diagonally down. At the bottom is a paddle that can move left and right

controlled by the arrow keys.When the ball hits the bottom, but misses the paddle, it disappears from the game. When the ball hits the paddle, it bounces back and continues diagonally up to the right. When the ball exits the playing field, a new ball comes into play.

Classes needed:

Ball - a Posn and the direction in which the ball moves (up or down)

Paddle - a Posn

PongWorld - contains one Ball and one Paddle, has a fixed width and height

The code in the file *pong-game-skeleton.ijava* defines the classes that represent the ball, the paddle, and the world. (For now, we ignore the world).

The class *PongWorld* extends the class *World* in the teachpack. Therefore, we need to use *ProfessorJ Intermediate Language*.

4.4 Designing methods in the class *Ball*

1. Design the method *draw* that displays the ball on a canvas. The following code (that can be written within the *Examples* class shows how you can draw one circle:

```
import draw.*;
import colors.*;
import geometry.*;

class Examples{
  Examples() {}

  Canvas c = new Canvas(200, 200);

  boolean makeDrawing =
    this.c.show() &&
    this.c.drawDisk(new Posn(100, 150), 50, new Red());
}
```

The three *import* statements on the top indicate that we are using the code programmed by someone else and available in the libraries named *draw*, *colors*, and *geometry*. Open the *Help Desk* and look under the *Teachpacks* for the teachpacks for *How to Design Classes* to find out more about the drawing and the *Canvas*.

2. Design the method *moveBall* that moves the ball five pixels in its direction.

3. Design the method *bounce* that produces a ball after it bounced up to the right, and with its direction set to move up to the right.

4. Design the method *hitBottom* that determines whether the ball hit the bottom of the canvas of the given height.

5. Design the method *outOfBounds* that determines whether the ball is out of bounds of the canvas of the given width and height.

4.5 Designing methods in the class *Paddle*

1. Design the method *draw* that displays the paddle on a given canvas of the given height.

2. Design the method *movePaddle* that consumes a *String* and moves the paddle either left or right depending on the *String* it receives as argument. For now, ignore the requirement that the paddle stays within the bounds of the canvas. (You may add it later, once the program is working.)

3. Design the method *hitBall* that determines whether the paddle hit the given ball. For simplicity, just make sure that the distance between the center of the ball and the center of the top of the paddle is less than or equal to the radius of the ball. You may need to delegate the work to the class *Ball*.

4.6 Designing methods in the class *PongWorld*

1. Design the methods *draw* and *erase* that show the background, the ball, and the paddle. Make the background black.

2. Finally, add some interactions to your program, by letting the paddle move in response to the key events. Design the method *onKeyEvent* that consumes a *String* and moves the paddle left, or right by 5 pixels every time the user hits one of the corresponding arrow keys.

   Use the code in the program *WorldDemo.ijava* to figure out how to respond to the key events and to see how to run the program.

3. Design the method *onTick* as follows:

- If the ball is out of bounds, replace the ball with a new ball. Initially, start the new ball in the top left corner. When everything else is working, make the ball start on the left edge at a random height between the top and the middle.
- If the ball hit the paddle, replace the ball with a new one going in the opposite direction - and moved ahead.
  Hint: Recall the method bounce in the class Ball.
- Otherwise, just move the ball in its direction.
- Currently the world never ends. When all is working, think of what may be the appropriate end of the game (count the number of balls that were put into play, the number of balls that hit the paddle, or the number of elapsed ticks. Then modify the appropriate methods so that the world eventually ends.

Save all your work — the next lab may build on the work you have done here!

Quiz

5 All are equal, but some are more equal than others.

Goals:

Learn how to determine the equality of two objects in a Java program.

The definitions of all the classes are already provided. The classes include a method *translate* in the class *CartPt* and the method *move* in the remaining classes. Both methods consume the distance *dx* and *dy* by which the items should be moved or translated. Additionally, some sample data is also given. Your goal is to design the method *same* that determines whether the values of two objects are the same (according to our definition of *sameness*).

5.1 Equality of simple classes

We start with our class of CartPt. The class is defined as follows:

```
+---------------------------------+
| CartPt                          |
+---------------------------------+
```

```
| int x                           |
| int y                           |
+---------------------------------+
| CartPt translate(int dx, int dy) |
+---------------------------------+
```

The method *move* is defined as follows:

// translate the position of this point by the given dx **and** *dy*
CartPt translate(*int dx*, *int dy*){
  return new CartPt(*this.x* + *dx*, *this.y* + *dy*);
}

Our tests are designed as follows:

CartPt pt1 = *new CartPt*(20, 30);

boolean testMove = *check pt1.move*(−5, 8) *expect new CartPt*(15, 38);

Of course we know, that the *check* form compares the values of *pt1* and the new *CartPt* we specified as the expected result. To replace this test by our own, we need a method in the class *CartPt* that determines whether this point is the same as the given one.

// is this point the same as the given one?
boolean same(*CartPt that*){...}

Design this method.

5.2 Equality of classes with containment

We now want to see if two stars in our Shooting Stars program are the same. Here is the class diagram for the class *Star*:

```
+--------------------------+
| Star                     |
+--------------------------+
| CartPt loc               |
| int lifespan             |
+--------------------------+
| Star move(int dx, int dy) |
+--------------------------+
```

Design the method *same* **that determines whether two stars are the same.** We consider two stars to be the same if they are at the same location and have the same lifespan.

Rewrite the tests as follows (remember, as the star moves, it also decreases its lifespan):

CartPt pt1 = *new CartPt*(20, 40);
boolean testTranslate = *pt.translate*(3, −5).*same*(*new CartPt*(23, 35));

Star star = *new Star*(*this.pt1*, 9);
boolean testMove = *star.move*(3, −5).*same*(*new Star*(*new CartPt*(23, 35), 8);

Save all your work — the next lab may build on the work you have done here!